

# Survey on Combinatorial Register Allocation and Instruction Scheduling

Roberto Castañeda Lozano and Christian Schulte

SCALE, KTH Royal Institute of Technology & Swedish Institute  
of Computer Science, Sweden  
`{rcas,cschulte}@kth.se`

## Abstract

Register allocation and instruction scheduling are two central compiler back-end problems that are critical for quality. In the last two decades, combinatorial optimization has emerged as an alternative approach to traditional, heuristic algorithms for these problems. Combinatorial approaches are generally slower but more flexible than their heuristic counterparts and have the potential to generate optimal code. This paper surveys existing literature on combinatorial register allocation and instruction scheduling. The survey covers approaches that solve each problem in isolation as well as approaches that integrate both problems. The latter have the potential to generate code that is globally optimal by capturing the trade-off between conflicting register allocation and instruction scheduling decisions.

## 1 Introduction

Compiler back-ends take an intermediate representation (IR) of a program and generate assembly code for a particular processor. The main problems in assembly code generation are instruction selection, register allocation, and instruction scheduling. Register allocation assigns temporaries (program variables in the IR) to processor registers or to memory. Instruction scheduling reorders instructions to improve their throughput. Because of the increasing access latency gap between memory and processor registers, register allocation has a significant impact on the performance of the generated code [40, 67]. Instruction scheduling has a significant impact for in-order processors (which issue instructions in the order dictated by the assembly) and is of paramount importance for very long instruction word (VLIW) processors [25] (which are able to issue multiple, statically scheduled instructions in each execution cycle). Out-of-order processors (which re-schedule instructions in hardware) also benefit from compile-time

instruction scheduling, since their dynamic reordering algorithms have limited scope.

This paper surveys existing literature on combinatorial optimization approaches to register allocation and instruction scheduling. Traditional, heuristic approaches to these problems are complex, hard to adapt to new processors, and generate suboptimal code. Combinatorial optimization has emerged as an alternative approach that is generally slower but more flexible than its heuristic counterparts and has the potential to generate optimal code [60]. Furthermore, combinatorial optimization makes it possible to solve both problems in integration to generate code that is better than the combination of the optimal solutions to each individual problem.

The survey introduces a detailed taxonomy of the existing combinatorial register allocation and instruction scheduling approaches, which enables a critical comparison, illustrates the developments and trends in the area, and exposes problems that remain unsolved. To the best of our knowledge, it contributes the first review devoted to combinatorial approaches to register allocation and instruction scheduling. Available surveys of register allocation [47, 74, 70, 73], instruction scheduling [76, 3, 81], and integrated code generation [52] ignore or present only a very brief description of combinatorial approaches.

**Heuristic approaches.** Traditional back-ends apply heuristic algorithms to solve register allocation and instruction scheduling, as these are hard combinatorial problems for which optimal solving has been commonly considered computationally infeasible. Heuristic algorithms sacrifice optimal solutions in favor of low complexity by taking a sequence of greedy decisions based on local criteria. Common heuristic algorithms for register allocation include graph-coloring [12, 9, 30], puzzle-solving for irregular register architectures [69], and linear scan for just-in-time compilation [72]. A classic heuristic approach to instruction scheduling is list scheduling [76].

**Combinatorial approaches.** Given the combinatorial nature of register allocation and instruction scheduling and their significant impact on assembly code quality, numerous ways of approaching them with combinatorial optimization techniques have been proposed starting in the early 1990s. Some factors underpinning this trend are: recent progress in optimization technology, improved understanding of the structure of the code generation problems, and increasing complexity and diversity of processors, particularly for embedded systems. The most explored combinatorial optimization techniques for code generation are Integer Programming, Constraint Programming, and Partitioned Boolean Quadratic Programming. These techniques generally target the same types of problems but often have complementary strengths.

**Integrated code generation.** Register allocation and instruction scheduling are interdependent problems: the solution to one of them affects the other [32].

Aggressive instruction scheduling often leads to programs that require more registers to store their temporaries which makes register allocation more difficult. Conversely, doing register allocation in isolation imposes a partial order among instructions which makes instruction scheduling more difficult. Given the interdependencies between problems, a fair amount of research has been devoted to study combinatorial approaches that solve register allocation and instruction scheduling in integration. Integrated combinatorial approaches can deliver globally optimal solutions by capturing the trade-offs between both problems. Integrated combinatorial approaches are better suited for this problem than their heuristic counterparts, since the underlying combinatorial models of each code generation problem are composable as Section 5 shows.

**Outline of the paper.** Section 2 describes the most popular combinatorial optimization techniques for code generation. Sections 3 and 4 present combinatorial approaches to register allocation and instruction scheduling in isolation. Section 5 discusses integrated combinatorial approaches to both problems, and the paper concludes with Section 6.

## 2 Combinatorial Optimization Approaches

Combinatorial optimization is a collection of *formal* and *complete* techniques to model and solve hard combinatorial problems. Combinatorial optimization techniques are *formal* since they capture combinatorial problems with a generic, formal modeling language. They are *complete* since they automatically explore the full solution space and guarantee to eventually find the optimal solution to a combinatorial problem (or prove that there is none). This section presents the most popular combinatorial techniques for code generation: Integer Programming (IP), Constraint Programming (CP), and Partitioned Boolean Quadratic Programming (PBQP). Search-based techniques which are not formal or complete lie outside of the scope of this survey. Examples of these are genetic algorithms [77] and dynamic programming [16].

Section 2.1 presents the modeling languages of each combinatorial optimization technique; Section 2.2 describes their main solving mechanisms.

### 2.1 Modeling

Combinatorial models consist, regardless of the particular optimization technique used, of the following elements:

**Variables** represent the decisions that are combined to form solutions. Variables can take values from different domains (for example, integers  $\mathbb{Z}$  or Booleans  $\{0, 1\}$ ). The variables in a model are represented here as  $x_1, x_2, \dots, x_n$ .

**Constraints** are relations over the variables that must hold for any solution. The structure of the constraints depends on the applied technique.

**Objective function** is an expression on the model variables to be minimized by the solver.

technique	variables	constraints	objective function
IP	$x_i \in \{0, 1\}$	$\sum_{i=1}^n a_i x_i \leq b$ ( $a_i, b, c_i \in \mathbb{Z}$ constant coefficients)	$\sum_{i=1}^n c_i x_i$
CP	$x_i \in D$	any $r(x_1, x_2, \dots, x_n)$ ( $D \subset \mathbb{Z}$ finite integer domain)	any $f(x_1, x_2, \dots, x_n)$
PBQP	$x_i \in D \subset \mathbb{Z}$	-	$\sum_{i=1}^n c(x_i) + \sum_{i,j=1}^n C(x_i, x_j)$ ( $c(x_i)$ cost of $x_i$ ; $C(x_i, x_j)$ cost of $x_i \wedge x_j$ )

Table 1: Typical modeling elements for different techniques.

**Integer Programming (IP).** IP [68] is a special case of Linear Programming where the variables range over integers. This survey assumes that IP only considers 0-1 variables for simplicity, since the vast majority of applications to code generation is limited to such a domain. The constraints and objective function of IP models are linear as shown in Table 1.

**Constraint Programming (CP).** CP [82] models can be seen as a generalization of IP models where the variables take values from a finite integer domain  $D \subset \mathbb{Z}$  (which can emulate 0-1 variables if  $D = \{0, 1\}$ ), and the constraints and the objective function are expressed by general relations and expressions on the problem variables. Often, such relations are formalized as *global constraints* that express commonly occurring substructures involving several variables.

**Partitioned Boolean Quadratic Programming (PBQP).** PBQP [83] is a special case of the Quadratic Assignment Problem [58]. As for CP, variables range over a finite integer domain  $D \subset \mathbb{Z}$ . However, PBQP models do not explicitly formulate constraints. Instead, each single variable assignment  $x_i$  is given a cost  $c(x_i)$  and each pair of variable assignments  $x_i \wedge x_j$  is given a cost  $C(x_i, x_j)$ . Pairs of assignments can be then forbidden by setting their cost to a practically infinite value. The objective function is formed by the cost of each individual assignment and the cost of each pair of assignments as shown in Table 1.

## 2.2 Solving

**Integer Programming.** IP solvers proceed by exploiting *linear relaxations* and *branch-and-bound search*, as well as other techniques. For an overview of IP, see for example [68]. As a first step, an optimal solution to a relaxed Linear Programming (LP) problem, where the variables can take any real value between 0 and 1, is computed. LP relaxations are enabled by the linear nature of the IP models and can be computed efficiently in practice. If all the variables in the solution to the LP problem are integral, this optimal solution also applies to the original IP problem. Otherwise, search decomposes the problem into two subproblems in which a non-integral variable is assigned the values 0 and 1 and the process is repeated. LP relaxations provide lower bounds on the value of the objective function which are applied to prove optimality. Valid solutions found during solving provide upper bounds which are applied by branch-and-bound search to reduce the search space.

**Constraint Programming.** CP solvers proceed by interleaving *constraint propagation* and *branch-and-bound search*. More information on CP can be found in [82]. Constraint propagation discards values for variables that cannot be part of any solution. When no further propagation is possible, search tries several alternatives on which constraint propagation and search is repeated. As for IP solving, valid solutions found during solving are exploited by branch-and-bound search to reduce the search space. Constraint propagation is essential to reduce the search space. Global constraints play a key role in propagation as they are associated to particularly effective propagation algorithms.

**Partitioned Boolean Quadratic Programming.** State-of-the-art PBQP solvers proceed in two phases: *reduction* and *back-propagation* [38]. Reduction transforms the original problem iteratively until the objective function becomes trivial to handle (that is, only the costs of individual assignments  $c(x_i)$  remain). This is achieved by applying a set of reduction rules. Special rules that reduce the problem in polynomial time while guaranteeing optimality of the final solution are prioritized. If none of these rules apply, reduction continues either by applying branch-and-bound search to preserve optimality, or by taking a problem-dependent, heuristic decision. After reducing the problem to a trivial one, back-propagation reconstructs the original problem by assigning values to variables to minimize the cost of the final solution.

## 3 Register Allocation

Register allocation takes as input a function where instructions of a particular processor have already been selected. Functions are typically represented by their control-flow graph (CFG). A basic block in the CFG consists of a straight-line sequence of *instructions* of maximum length. Instructions use and define

*temporaries*. Temporaries are storage locations holding values corresponding to program variables after instruction selection.

A *program point* is located between two consecutive statements. A temporary  $t$  is *live* at a program point if  $t$  holds a value that might be used in the future. The *live range* of a temporary  $t$  is the set of program points where  $t$  is live. Two temporaries *interfere* if their live ranges overlap.

**Register allocation and assignment.** Register allocation assigns temporaries to either processor registers or memory. Register assignment maps non-interfering temporaries to the same register to improve register utilization and avoid the use of memory as much as possible.

**Spilling.** In general, optimal register utilization does not guarantee the availability of enough processor registers and some temporaries must be *spilled* (that is, stored in memory). Spilling a temporary requires the insertion of store and load instructions to move its value to and from memory. The selection and placement of these instructions has considerable impact on the efficiency of the generated code. *Spill code optimization* reduces the cost of a spill by optimizing the placement of its corresponding store and load instructions.

**Coalescing.** The input program may contain temporaries related by *copies* (operations that replicate the value of a temporary into another). Copy-related temporaries that do not interfere can be *coalesced* (assigned to the same register) in order to discard the corresponding copies and thereby improve efficiency and reduce code size.

**Live-range splitting.** In some cases, it is desirable to allocate a temporary  $t$  to different locations during different parts of its live range. This is achieved by *splitting*  $t$  into a temporary for each part of the live range that is to be allocated to a different location.

**Packing.** Each temporary  $t$  has a certain bit width which is determined by  $t$ 's source data type. Many processors allow temporaries of small widths to be assigned to different parts of a physical register of larger width. This characteristic is commonly known as *register aliasing*. For example, Intel's x86 [45] combines pairs of 8-bits registers (AH, AL) into 16-bit registers (AX). Packing non-interfering temporaries into the same physical register is key to improving register utilization.

**Rematerialization.** In processors with a very limited number of registers can be sometimes beneficial to recompute (that is, *rematerialize*) a reused value rather than occupying a register until its later use, or spilling.

**Multiple register banks.** Some processors include multiple processor banks clustered around different types of functional units, which often leads to alternative temporary allocations. To handle effectively these architectures, register allocation needs to take into account the cost of allocating a temporary to different register banks and moving its value across them.

**Scope.** *Local* register allocation deals with one basic block at a time, spilling all temporaries that are live at basic block boundaries. *Global* register allocation considers entire functions, yielding better code as temporaries can be kept in the same register across basic blocks.

### 3.1 Optimal Register Allocation (ORA)

The most prominent combinatorial approach to global register allocation is presented in 1996 by Goodwin and Wilken [33]. The approach, called *Optimal Register Allocation* (ORA), is based on an IP model that captures the full range of register allocation subproblems (see Table 2 for a summary of the characteristics of each approach). Goodwin and Wilken’s ORA demonstrated, for the first time, that combinatorial global register allocation is feasible – although much slower than heuristic approaches.

The ORA allocator derives an IP model in several steps. First, a *Live Range Graph* (LRG) is constructed for each temporary  $t$  and register  $r$  in the program (Goodwin and Wilken use the terms *symbolic registers* and *symbolic register graphs* to refer to temporaries and LRGs [33]). Then, different program points of the LRG are annotated with binary decisions that correspond to 0-1 variables in the IP model and linear constraints (called *conditions* in [33]) involving groups of decisions.

**Model.** The model includes four main groups of variables to capture different subproblems: *register assignment variables*  $\text{def}(t, r)$ <sup>1</sup>,  $\text{use-cont}(t, r)$ , and  $\text{use-end}(t, r)$  indicate whether temporary  $t$  is assigned to  $r$  at each definition and use (use-cont and use-end reflect whether the assignment continues after the use or it is ended); *spilling variables*  $\text{store}(t, r)$ ,  $\text{cont}(t, r)$ , and  $\text{load}(t, r)$  indicate whether temporary  $t$  which is assigned to register  $r$  is stored in memory, whether the assignment to  $r$  continues after a possible store, and whether  $t$  is loaded from memory to  $r$ ; *coalescing variables*  $\text{elim}(t, t', r)$  indicate whether the copy from  $t$  to  $t'$  is eliminated by assigning  $t$  and  $t'$  to  $r$ ; and *rematerialization variables*  $\text{remat}(t, r)$  indicate whether  $t$  is rematerialized into  $r$ .

The model includes linear constraints on these variables to enforce that at each program point, each register holds at most one temporary; each temporary is assigned to a register at each program point where it is defined and used; each temporary is assigned the same register where its live ranges are merged

---

<sup>1</sup>Note that the original variable names and constraints in the publications cited by this survey are sometimes altered or simplified for the sake of clarity and comparability of different approaches.

at the join points of the CFG; and an assignment of temporary  $t$  to a register that holds right before a use must be conserved until the program point where  $t$  is used. Other constraints to capture spilling, coalescing, and rematerialization are listed in [33].

The objective function minimizes the total overhead of decisions reflected on the spilling, coalescing, and rematerialization variables.

**Solving.** Goodwin and Wilken use an off-the-shelf commercial IP solver to derive register allocations and compare the results against those of GCC’s register allocator for a Hewlett-Packard PA-RISC processor [48]. The experiments reveal that in practice register allocation problems have a relatively low average complexity, and functions of hundreds of instructions can be optimally solved in a time scale of minutes.

**Model extensions for irregular processors.** The results of Goodwin and Wilken encouraged further research based on the ORA approach. In 1998, Kong and Wilken present a set of extensions [57] to the original ORA model, including register packing and handling multiple register banks, to deal with irregularities in register architectures. The extensions are complete enough to be able to deal with Intel’s x86 architecture [45], which presents a fairly irregular register file. The results improve drastically the code quality of GCC – the extended ORA approach reduces the execution time overhead due to register allocation by 61% on average. An interesting finding is that IP solving, as opposed to heuristic approaches, becomes easier in the face of less registers and more irregularities – the solving times of Goodwin and Wilken are reduced by two orders of magnitude.

**Faster ORA model.** The (still large) solving time gap between the ORA and heuristic approaches is reduced by Fu and Wilken in 2002 [27]. Fu and Wilken identify numerous conditions under which decisions are *dominated*. A 0-1 decision (variable) is *dominated* if there always exists an equal or better solution where the decision is not taken [14]. Fu and Wilken remove dominated spilling variables from the model to decrease its complexity.

Experiments where the reduced model is compared to the original one by Goodwin and Wilken show how the IP solving process is sped-up by four orders of magnitude: two due to increased computational power and algorithmic improvements in the IP solver and two due to the removal of dominated variables and their corresponding constraints. The improvements allow solving 98.5% of the functions in the SPEC92 integer benchmarks optimally with a time limit of 1024 seconds.

### 3.2 Partitioned Boolean Quadratic Programming (PBQP)

Scholz and Eckstein propose in 2002 an alternative combinatorial approach to ORA that models register allocation as a PBQP problem [83]. The simplicity



with which register allocation can be reduced to a PBQP problem and the availability since 2008 of a production-quality implementation<sup>2</sup> within the LLVM [62] compiler infrastructure have made PBQP one of the most popular optimization techniques for code generation. However, the simplicity of PBQP also comes with limitations – the range of subproblems that can be captured is narrower than that of the more general ORA approach. In particular, the PBQP approach does not support live-range splitting (see Table 2).

**Model.** In contrast to the rather complex model of the ORA approach, the PBQP model features a single class of variables  $a(t)$  giving the register to which temporary  $t$  is assigned. The decision to spill  $t$  to memory is captured by including a special spilling register  $sp$  to the domain of each variable  $a(t)$ . That is, the assignment  $a(t) = sp$  indicates that  $t$  is spilled to memory. The inclusion of spilling registers in the domains of the assignment variables guarantees the existence of a trivial solution where all temporaries are spilled. Similarly, a special rematerialization register  $rm$  is added to the domain of each variable  $a(t)$  to introduce the possibility of rematerializing  $t$  [37, Chapter 4]. For the sake of solving, Scholz and Eckstein decompose each variable  $a(t)$  into a collection of alternative Boolean variables  $\{x(t, sp), x(t, R0), x(t, R1), \dots, x(t, Rm)\}$ , but the formulation with finite integer variables is preferred here for simplicity. As is characteristic of PBQP models (see Table 1), constraints are expressed as practically infinite costs in individual assignments  $c(a(t))$  and pairs of assignments  $C(a(t), a(t'))$ . Individual costs  $c(a(t))$  are used to forbid assignments of temporaries to registers that do not belong to supported register classes, take into account the overhead of spilling or rematerializing  $t$ , and take into account the benefit of coalescing  $t$  with pre-assigned registers. Costs of pairs of assignments  $C(a(t), a(t'))$  are used to forbid assignments of interfering temporaries to the same register, forbid register assignments of interfering temporaries to aliased registers; and take into account the benefit of coalescing  $t$  and  $t'$ . The objective function minimizes the total cost of individual and pairs of assignments, effectively enforcing the constraints that would lead to a practically infinite cost.

**Solving.** Scholz and Eckstein devised a custom heuristic solver that follows the procedure explained in Section 2.2 and an optimal solver that applies exhaustive enumeration (as opposed to branch-and-bound) search when no reduction rule applies. Results for a digital signal processor (DSP) show that the PBQP approach can deliver up to 13.6% faster programs than traditional, graph coloring-based heuristics. At the same time, the results show a significant quality gap between the solutions delivered by the heuristic and the optimal PBQP solver. Further experimentation on the optimal PBQP solver by Hirnschrott *et al.* supports the conclusions of Scholz and Eckstein for DSPs with different number of registers and instruction operands [41].

---

<sup>2</sup> <http://permalink.gmane.org/gmane.comp.compilers.llvm.devel/34822>

**Improved PBQP solver.** In 2006, Hames and Scholz [38] contribute improved reduction heuristics and a branch-and-bound search mechanism to the PBQP solver. The heuristics apply ideas from graph-coloring register allocation to choose the temporary that is selected for reduction when no rule applies. As opposed to the scheme of Scholz and Eckstein, temporaries are only committed to particular registers during the *back-propagation* phase.

Hames and Scholz’s experiments for Intel’s x86 architecture [45] show that Scholz and Eckstein’s heuristic perform poorly for general-purpose code and processors such as x86 which are more regular than typical DSPs, the branch-and-bound PBQP solver solves 97.4% of the SPEC2000 functions over 24 hours, and the new heuristic PBQP solver is on average 2% from the optimal solutions and almost in all cases generates better solutions than the graph coloring-based heuristics.

### 3.3 Progressive Register Allocation (PRA)

Koes and Goldstein introduce in 2005 a *progressive* approach to register allocation [55] which is extended and improved in 2006 [56]. An ideal (combinatorial) progressive solver should 1) deliver reasonable solutions quickly, 2) improve them if more time is allowed, and 3) find the optimal one if enough time is available. Although both the ORA and the PBQP approach can also potentially behave progressively, in 2005 none of them were able to meet the first condition.

Koes and Goldstein propose to model register allocation as a multi-commodity network flow (MCNF) problem [2], which can be seen as a special case of an IP problem. The MCNF problem consists of finding a flow of multiple commodities through a network such that the cost of flowing through all arcs is minimized and the flow capacity of each arc is not exceeded.

The reduction of register allocation to MCNF is intuitive: each commodity corresponds to a temporary which flows through the function instructions. This model can express detailed allocations and accurately take into account their cost. Furthermore, the approach can leverage well-understood solving techniques available for network problems to yield progressive solving behavior. On the other hand, the flow abstraction can cope with neither coalescing nor register packing, and since the network’s flow must be conserved, each temporary can only be allocated to a single location at each program point.

**Model.** As mentioned above, each commodity in the network flow corresponds to a temporary. For each program point and storage location (registers and memory), a node is added to the network. The flow of a temporary  $t$  through the network determines how  $t$  is allocated. As for the PBQP model, a single class of variables is defined:  $a(t, (i, j))$  indicates whether the temporary  $t$  flows through the arc  $(i, j)$  where  $i, j$  represent storage locations at particular program points. Compared to the PBQP model, the additional variable dimension corresponding to program points permits expressing more detailed register allocations capturing live-range splitting. The model includes linear constraints to enforce that only one temporary flows through each register node (called *bundle*

*constraints* by Koes and Goldstein [56]), and flow is conserved through the nodes (the same amount of flow that enters a node exits it). The objective function minimizes the arc traversal cost for all flows. The cost of an arc  $c((i, j))$  reflects the cost of moving a temporary from the source location  $i$  to the destination location  $j$ : if  $i$  and  $j$  correspond to the same locations, no cost is incurred; if  $i$  and  $j$  correspond to different registers,  $c((i, j))$  is the cost of a register-to-register move instruction; and if one of  $i$  and  $j$  corresponds to memory and the other to a register,  $c((i, j))$  is the cost of a memory access instruction.

**Solving.** Although The PRA model is a valid IP formulation and can be thus solved by a regular IP solver, Koes and Goldstein design a dedicated solving scheme with the goal of delivering a more progressive behavior. The solving scheme is mainly based on a Lagrangian relaxation, a general IP technique that, similarly to PBQP models, replaces hard constraints by terms in the objective function that penalize their violation. Relaxing the bundle constraints allows to decompose the problem into one subproblem for each temporary. Each subproblem is then solved with an iterative optimization method based on computations of shortest paths (which correspond to least-cost individual register allocations). The process is iterated until the bundle constraints are no longer violated, at which point the solution is an optimal register allocation. At the end of each iteration, heuristics guided by information obtained from the Lagrangian relaxation deliver progressively improving solutions.

Koes and Goldstein compare their progressive solver with GCC’s register allocator and a commercial IP solver. Experiment results with functions from different benchmarks show that the PRA approach is indeed more progressive than other solving techniques such as standard IP: it delivers first solutions in a time that is competitive with traditional heuristic approaches and generates optimal solutions for 83.5% of the functions after 1000 iterations. The optimal solutions yield in average a code size reduction of 6.8% compared to a heuristic graph-coloring approach.

### 3.4 Other Approaches

A recent approach that is radically different from those discussed above is that of Krause [59]. Krause exploits the property that most programming languages yield structured programs [86] to obtain optimal register allocations in polynomial time, where the polynomial degree is given by the (constant) number of registers of a processor. Krause’s approach applies a dynamic programming (DP) solving strategy, where locally optimal decisions are extended to globally optimal ones. The approach can deal with most of the register allocation subproblems (see Table 2), and experiments show that it is best suited for processors with few registers.

Other approaches have focused on decomposing the vast amount of subproblems included in register allocation and solving each of the resulting problems to optimality. This pragmatic strategy aims at improving code generation time while still achieving near-optimal (but not globally optimal) solutions. In 2001,

Appel and George propose to solve spilling (including spill code optimization and live-range splitting) optimally with IP and to resort to heuristic algorithms for register assignment and coalescing [5]. Their experiments demonstrate that the approach scales better than the full-fledged ORA solver (as to be expected since less subproblems are considered at the same time) and improves the speed of the code generated by a heuristic register allocator by 9.5%. In 2009, Ebner *et al.* introduce a more structured IP approach to the same problem as Appel and George in which spilling, spill code optimization, and live-range splitting are modeled as a minimum cut problem with capacity constraints [18]. A node in the minimum-cut network corresponds to a temporary at a particular program point. A solution corresponds to a cut in the network where one partition is allocated to memory and the other to processor registers. The problem is solved both with a dedicated solver based on a Lagrangian relaxation (where the capacity constraints are relaxed) and a commercial IP solver. Interestingly, the experiments show that the IP solver performs better than all but the simplest version of the dedicated solver, delivering optimal solutions in less solving time. Finally, Colombet *et al.* introduce in 2011 an alternative IP approach [15] that additionally captures rematerialization, overcomes some prior limitations (including the inability to allocate a temporary to memory and a processor register locations simultaneously), and handles programs in Static Single Assignment (SSA) form [17]. This program form, where temporaries are only defined once, has proved to have useful properties for register allocation [36]. Colombet *et al.*'s experiments with the EEMBC benchmarks for a VLIW processor show that the estimated cost (that is, the optimal value to the objective function) is improved on average by around 40 % compared to the model of Appel and George, where the addition of rematerialization accounts for approximately half of the improvement. An interesting finding is that these substantial estimated improvements only correspond to modest runtime improvements, due to interactions with the later instruction scheduling and bundling stage.

Together with their optimal spilling approach, Appel and George launched in 2000 the *Optimal Coalescing Challenge*<sup>3</sup>, a benchmark for the second phase of their decomposed approach to register allocation [5]. In 2007, Grund and Hack present an optimal coalescing approach based on IP [35] that solves 90.7% of the coalescing problems in the Optimal Coalescing Challenge to optimality. The approach reduces the input problem by preassigning temporaries to registers whenever it is safe and exploits the structure of the interference graph to derive additional constraints that speed-up solving.

## 4 Instruction Scheduling

Instruction scheduling maps program instructions to basic blocks and issue cycles within the blocks. A valid instruction schedule must satisfy the dependencies among instructions and cannot exceed the capacity of any processor resource such as data buses and functional units. Typically, instruction scheduling aims

---

<sup>3</sup><http://www.cs.princeton.edu/~appel/coalesce>

approach	TC	SC	SP	RA	CO	SO	RP	LS	RMMB	SZ
ORA [33, 57, 27]	IP	global	✓	✓	✓	✓	✓	✓	✓	~ 2000
PBQP [83, 41, 38]	PBQP	global	✓	✓	✓	-	✓	-	✓	?
PRA [55, 56]	IP	global	✓	✓	-	✓	-	✓	✓	?
Krause [59]	DP	global	✓	✓	✓	✓	✓	✓	✓	?
Appel and George [5]	IP	global	✓	-	-	✓	-	✓	-	~ 2000
Ebner <i>et al.</i> [18]	IP	global	✓	-	-	✓	-	✓	-	?
Colombet <i>et al.</i> [15]	IP	global	✓	-	-	✓	-	✓	✓	?
Grund and Hack [35]	IP	global	-	✓	✓	-	-	✓	-	?

Table 2: Combinatorial register allocation approaches: technique (TC), scope (SC), spilling (SP), register assignment (RA), coalescing (CO), spill code optimization (SO), register packing (RP), live-range splitting (LS), rematerialization (RM), multiple register banks (MB), and size of largest optimal solution in number of instructions (SZ).

at minimizing the *makespan* of the computed schedule – the number of cycles it takes to execute the schedule. All models presented in this section, unless stated otherwise, have makespan minimization as their objective function.

**Dependencies.** Flow of data and control cause dependencies among instructions. The dependencies in each basic block of a program form a *dependency graph* (DG) where vertices represent instructions and an arc  $(i, j)$  indicates that instruction  $j$  depends on instruction  $i$ . Arcs in a DG are often labeled with the latency  $l(i, j)$  between the corresponding instructions. The latency dictates the minimum amount of cycles that must elapse between the issue of the two instructions. For convenience, combinatorial instruction scheduling approaches usually add artificial *entry* and *exit* instructions to the DG that precede and succeed all other instructions.

**Resources.** Instructions share limited processor resources such as functional units and data buses. The organization of hardware resources varies widely among different processors and profoundly affects the complexity of instruction scheduling. Resources are classified as *single-* or *multi-capacity* depending on the number of instructions that can access them at the same time. Instructions can use either one resource each (*single usage*) or multiple resources (*multiple usage*). Resource usage can be *uni-* or *two-dimensional* depending on whether its duration is one or longer than one. Finally, resource usage is either *synchronous* if usage by an instruction  $i$  starts at the issue cycle of  $i$  or *asynchronous* otherwise.

**Instruction bundling.** Traditional RISC processors are *single-issue*: they can issue only one instruction at each clock cycle. Modern processors are usually *multiple-issue*: they can issue several instructions every clock cycle. To exploit

this capability, in-order multiple-issue processors such as VLIW processors require the compiler to perform *instruction bundling*, combining instructions into bundles that are compatible with the issue restrictions of the processor. Such restrictions are often caused by underlying resource constraints which can be handled analogously to the constraints imposed by other processor resources.

**Scope.** Unlike register allocation which is almost always handled globally, the instruction scheduling literature covers different problem scopes. This can be classified into three levels, where each level can potentially deliver better code at the expense of increasing the complexity of scheduling: *local instruction scheduling* schedules instructions within basic blocks in isolation, under the assumption that each instruction is already placed in a certain basic block; *regional instruction scheduling* considers collections of basic blocks with a certain CFG structure; and *global instruction scheduling* considers entire functions. Besides assigning each instruction  $i$  to an issue cycle within its block, regional and global instruction scheduling must decide the basic block into which  $i$  is placed.

## 4.1 Local Instruction Scheduling

**Early approaches.** An early combinatorial approach to local instruction scheduling is introduced by Arya in 1985 [6]. This approach uses IP to compute optimal schedules for single-issue vector processors such as the early Cray-1. Arya’s model includes non 0-1 integer variables  $T(i)$  and  $E(i)$  representing the issue and execution cycle of each instruction  $i$ , which are not necessarily equal in the targeted processor. Both dependency and resource constraints are modeled with inequalities on pairs of cycle variables. This simple model for resource constraints is possible as all resources are of single-capacity. Thus, two instructions  $i, j$  using the same resource simultaneously can be prevented by a constraint where either  $i$  precedes  $j$  or vice versa. Arya’s approach includes an adaptation to single-basic block loops based on partial unfolding. Three experiments with a commercial IP solver on basic blocks of up to 36 instructions show that the approach is feasible and improves hand-optimized code significantly.

In 1991, Ertl and Krall introduce the first local instruction scheduling approach based on constraint programming [23]. Ertl and Krall use *constraint logic programming* [46] which embeds constraint propagation techniques into logic programming. Their model targets single-issue, pipelined RISC processors where the resource consumption of an instruction must be considered at each stage of the pipeline. The model includes, for each instruction  $i$  and pipeline stage  $k$ , a finite integer domain variable  $s(i, k)$  representing the cycle in which  $i$  resides in  $k$ . Dependencies among instructions are modeled as in Arya’s approach. Equality constraints link the different stage cycle variables of an instruction. To enforce that only one instruction at a time resides in each pipeline stage  $k$ , an *alldifferent* constraint [79] forcing all variables

$\{c(i_1, k), c(i_2, k), \dots, c(i_n, k)\}$  to take different values is used. Experiments on a few programs for the Motorola 88100 processor show that the approach improves 8.5% and 19% of the basic blocks scheduled by the Harris C compiler and GCC 1.3 respectively. However, the richness of Ertl and Krall’s resource model comes at the cost of low scalability: the experiments show that the approach cannot handle larger problems than Arya’s scheduler despite the six year gap between both publications.

The first combinatorial approach to instruction scheduling and bundling for multiple-issue processors is introduced by Leupers and Marwedel in 1997 [63]. Their model is inspired by early work on IP-based scheduling for high level synthesis [43, 29] and designed to target irregular DSPs. Among the additional challenges presented by such processors, the model incorporates *alternative instruction versions* and *side effect handling*. Alternative instruction versions (called *alternative encodings* by Leupers and Marwedel) supports the selection of different instruction implementations, each using different resources, for an instruction in the input program. Side effect handling supports inserting and scheduling *no-operation instructions* that inhibit writes to certain registers; and accounts for pairs of instructions that must be scheduled together if a particular version of one of them is chosen. To schedule instructions in a basic block and take into account the particularities of DSPs, Leupers and Marwedel introduce an IP model with two types of 0-1 variables: *scheduling variables*  $s(v, i, k)$  indicate whether version  $v$  of instruction  $i$  is issued in cycle  $k$ ; and *no-operation variables*  $n(r, k)$  indicate whether register  $r$  is inhibited by a no-operation in cycle  $k$ . The model includes constraints to ensure that each instruction is implemented and scheduled once, dependencies are satisfied, instruction versions that use the same resource are not issued in the same cycle, and no-operations are inserted to prevent destroying live data stored in registers. Leupers and Marwedel present some experimental results on a few signal-processing basic blocks for the TMS320C25 and Motorola DSP56k processors, reporting optimal results for basic blocks of up to 45 instructions.

**Modern approaches.** In 2000, a publication by Wilken *et al.* [88] triggers a new phase in the combinatorial instruction scheduling research. This phase is characterized by a higher ambition on scalability where the goal is to solve the largest basic blocks (containing thousands of instructions) from established benchmarks such as SPEC95 and SPEC2000. The focus shifts from being able to deal with highly irregular architectures to understanding and exploiting the structure of the DG (the main structure of instruction scheduling), and improving the underlying solving techniques by applying problem-specific knowledge.

Wilken *et al.* use a restricted form of the IP model by Leupers and Marwedel introduced above. The model contains scheduling variables  $s(i, k)$  indicating whether instruction  $i$  is issued in cycle  $k$ , and single-scheduling and dependency constraints similar to those of Leupers and Marwedel. The main contributions of Wilken *et al.* are DG transformations and IP solving techniques that make the original problem easier to solve. These contributions assume single-issue

processors although the model itself can be generalized to deal with multiple-issue processors. The contributed DG transformations include: *partitioning* to decompose DGs while preserving optimality, *redundant dependency elimination* to discard dependencies that do not affect the solution but incur overhead in the IP formulation, and *region linearization* to enforce dominating schedules in specific components of the DG. The devised solving techniques include: *iterative range reduction* and *instruction probing* to iteratively tighten the instruction cycle bounds and discard scheduling variables, *dependency and spreading cuts* to correct and tighten invalid linear relaxations, and *algebraic simplification* to reduce the number of variables involved in dependency constraints. Experiments on an ideal processor with latencies of up to three cycles show that the improvements by Wilken *et al.* allow to schedule all basic blocks from the SPEC95 floating-point benchmarks (containing up to 1000 instructions) to optimality.

Shortly after the publication of Wilken *et al.*, Van Beek and Wilken introduce a CP approach targeting the same processor model [87]. The model contains finite integer scheduling variables  $s(i)$  representing the issue cycle of each instruction  $i$ . Dependency constraints are directly expressed as inequalities on the scheduling variables. Similarly to Ertl and Krall [23], an *alldifferent* constraint [79] is used to force a single issue in each cycle. The model also contains two types of *implied constraints* (logically redundant constraints that increase the amount of propagation and thus reduce the search space): *distance constraints*, which impose a minimum issue distance among the boundary instructions of the regions as defined by Wilken *et al.*; and *predecessor/successor constraints*, which enforce lower/upper bounds on the scheduling variable domains of instructions with multiple predecessors/successors (these constraints can be seen as an application of *edge finding*, a general constraint propagation algorithm [7]). Van Beek and Wilken demonstrate, using a custom-made solver, that the CP approach is significantly faster (22 times) than the original IP approach for the same basic blocks from the SPEC95 floating-point benchmarks.

In 2006, Heffernan and Wilken [39] contribute two further DG transformations which generalize to multiple-issue processors: *superior nodes* and *superior subgraphs*. The first transformation identifies pairs of independent instructions in the DG such that relating them by an artificial dependency preserves the optimality of the original scheduling problem. The second transformation is a computationally heavier generalization that identifies pairs of isomorphic subgraphs in the DG such that their instructions can be also optimally related by artificial dependencies. The transformations are applied iteratively as their outcome might expose further transformation opportunities. Heffernan and Wilken show that the upper and lower makespan bounds of the basic blocks in the SPEC2000 floating-point benchmarks are significantly reduced after the transformations are applied, particularly for single-issue processors.

Malik *et al.* (including Van Beek) introduce in 2008 a CP approach that combines the above ideas to generalize to multiple-issue processors while remaining scalable [65]. The model is an extension of Van Beek and Wilken's model where single-issue *alldifferent* constraints are replaced by the more general *global cardinality* constraints [80]. A *global cardinality* constraint for each



processor resource limits the amount of instructions of the same type that can be issued in the same cycle. The use of this constraint enables a compact model where the individual resource used by each instruction does not need to be tracked with additional variables and constraints. Furthermore, the model contains four types of implied constraints: *distance* and *predecessor/successor constraints* generalized to multiple-issue processors, *safe pruning constraints* which discard suboptimal schedules with idle cycles, and *superior subgraph constraints* (called *dominance constraints* by Malik *et al.*) which apply the transformation introduced by Heffernan and Wilken as constraint propagation. Unlike all previous modern approaches, Malik *et al.* experiment on a multiple-issue processor with a bundle width varying from 1 to 6 instructions and a latency model similar to that of the PowerPC architecture. The results using a custom solver for the full SPEC2000 benchmark show that all but a few of the basic blocks (up to 2600 instructions) can be solved to optimality. Interestingly, the experiments show that the number of unsolved basic blocks does not increase with bundle widths beyond 2 instructions.

## 4.2 Regional Instruction Scheduling

Local instruction scheduling is well understood and, despite its theoretical complexity, has been shown in practice to be feasible for combinatorial optimization techniques. Unfortunately though, the scope of local instruction scheduling (single basic blocks) severely limits the amount of instruction-level parallelism that can be exploited, particularly for control-intensive programs. To overcome this limitation, regional instruction scheduling approaches have been proposed that operate on multiple basic blocks with a certain CFG structure. The most studied structure in the context of combinatorial instruction scheduling is the *superblock*. A superblock is a collection of consecutive basic blocks with a single entry point and possibly multiple exit points [44].

**Handling compensation code.** Moving an instruction from one basic block to another often requires the insertion of *compensation code* into yet another basic blocks to preserve the program semantics. In the case of superblocks, such a situation only arises when an instruction is moved after an exit point where the result of the instruction is required. The published regional instruction scheduling approaches based on combinatorial optimization avoid handling compensation code by either disregarding the additional cost of placements of instructions that require compensation code or just disallowing such placements. The latter can be achieved by adding dependencies from such instructions to their corresponding exit points, at the expense of limiting the scope for optimization.

**Superblock scheduling.** The first optimal approach to superblock scheduling is presented by Shobaki and Wilken in 2004 [84]. This approach is based on ad-hoc search (called *enumeration* by the authors) which is exhaustive and can

thus deliver optimal solutions in finite time. The approach is not discussed in detail here as the employed technique lacks the first step common to all combinatorial optimization approaches (that is, the definition of the problem with a generic, formal modeling language as discussed in Section 2.1). The structure of the problem definition by Shobaki and Wilken is similar to the model used by Heffernan and Wilken [39] and Malik *et al.* [65] for local instruction scheduling. The key difference with the local scheduling approaches lies in the objective function: while local scheduling aims at minimizing the makespan of a basic block, the objective function in superblock scheduling minimizes the *weighted makespan*. The weighted makespan of a superblock is the sum of the cycles in which each exit instruction is scheduled weighted by their exit likelihood:

$$\sum_{b \in B} \text{weight}(b) \times s(\text{exit}(b)) \quad (1)$$

where  $B$  is the set of basic blocks in the superblock and  $\text{exit}(b)$  gives the exit instruction of basic block  $b$ . This objective function is common to all published combinatorial optimization approaches for superblock scheduling.

In 2008, Malik *et al.* [64] present a CP approach that extends their local scheduler discussed above [65] to superblocks and a richer resource model. The basic model incorporates the following new elements to the local scheduling one: the objective function for superblocks discussed above (1); and additional variables and constraints to model two-dimensional resource usage and instructions that use all resources in their issue cycle. To scale to larger problem sizes, the model also incorporates superblock adaptations of *distance* and *superior sub-graph constraints*. Malik *et al.* employ a more sophisticated solving procedure than in previous approaches, involving *portfolio search* [31], where the level and complexity of propagation is increased as solving proceeds; *impact-based search* [78], where the solver selects first the variable that causes most propagation to try different search alternatives on; and *decomposition* [26], where the independent basic blocks that belong to larger superblocks are solved separately. A thorough experimental evaluation on the SPEC2000 benchmark for different variations of the PowerPC processor shows that the approach improves the results given by state-of-the-art heuristic algorithms and conserves the scalability demonstrated for basic blocks, despite the generalization to superblocks and the incorporation of a richer resource model.

**Trace scheduling.** Superblocks can be generalized into *traces* to increase further the scope for scheduling, at the expense of higher complexity in handling compensation code. A trace is a collection of consecutive basic blocks with possibly multiple entry and exit points [24]. Besides requiring compensation code due to exits (as for superblock scheduling), trace scheduling must insert compensation code when moving instructions before entry points.

The only published optimal approach for trace scheduling is due to Shobaki *et al.* [85]. This approach exploits the same technique as Shobaki and Wilken’s superblock scheduler and cannot thus be strictly seen as combinatorial optimization. Nevertheless, a brief discussion follows for completeness. The main

difference with the superblock approach by the same authors lies in the objective function. In trace scheduling, all possible execution paths through a trace must be considered. Also, it is desirable to account for the cost of inserting compensation code outside the trace. With these considerations in mind, Shobaki *et al.* propose the following objective function to be minimized:

$$kC + \sum_{p \in P} \text{weight}(p) \times (s(\text{exit}(b)) - s(\text{entry}(b))) \quad (2)$$

where  $C$  is a variable indicating the number of compensation instructions introduced in external blocks,  $k$  is a constant factor to control the trade-off between trace speed and size of the rest of the function, and  $P$  is the set of paths from some entry to some exit in the trace. The experimental results of Shobaki *et al.* for the SPEC2006 integer benchmarks and a UltraSPARC IV processor (with an instruction bundle width of four instructions) show that 91% of the traces (of up to 424 instructions) can be solved optimally, yielding a slight improvement of the weighted makespan (2.7% on average) and a noticeable reduction of the compensation code (15% on average).

**Software pipelining.** A fundamentally different form of regional scheduling is *software pipelining* [75, 61]. Software pipelining schedules the instructions of multiple iterations of a loop body simultaneously to increase the amount of available instruction-level parallelism. Typically, software pipelining is applied to inner-most loops consisting of single basic blocks although extensions are proposed to deal with larger structures [81, Section 20.4]. The output of software pipelining is a loop kernel with as short makespan as possible (called *initiation interval* in the context of software pipelining), together with a prologue and epilogue to the kernel where the instructions of the first and last iterations of the loop are scheduled. A comprehensive description of the problem is given by Allan *et al.* [3].

The most prominent combinatorial approach to software pipelining, based on IP, is due to Govindarajan *et al.* [34, 4]. As is common in software pipelining, the approach proceeds by trying to construct schedules of successively increasing initiation interval. The first schedule found is thus guaranteed to be *rate-optimal* (that is, of minimum makespan).

The basic IP model, presented in 1994 [34], includes two main groups of variables: *kernel scheduling variables*  $a(i, k)$  indicate whether instruction  $i$  is scheduled in cycle  $k$  within the loop kernel, and *offset scheduling variables*  $t(i)$  give the cycle in which the first instance of instruction  $i$  is issued. These variables are subject to linear constraints to enforce that: only one instance of each instruction is scheduled within the loop kernel, the capacity of the resources is not exceeded in any cycle, the data dependencies among instructions are satisfied, and the cycle of an instruction  $i$  in the kernel given by  $a(i, k)$  corresponds to its offset cycle  $t(i)$ . Furthermore, Govindarajan *et al.* propose extensions to deal with two-dimensional resources and multiple resource usage. Their experiments on 22 loops from the Livermore, SPEC92 and Whetstone benchmark

suites for a multiple-issue processor with a bundle width varying from one to six show that the optimal solution can be found in reasonable time (less than 30 seconds) for 91% of the loops. Although the maximum size of a loop is not indicated, the reported initiation intervals suggest that the approach can handle loops in the order of tens of instructions.

A year later, Altman *et al.* (including Govindarajan) present an extension of the model [4] which copes with asynchronous resource usage by keeping track of each individual use with 0-1 variables. Altman *et al.*'s experiments on 1066 loops from the Livermore, SPEC92, Linpack, NAS, and Whetstone benchmark suites for a variant of the PowerPC-604 show that, despite a significant increase in the complexity of the resource model, their extended approach can solve 75% of the loops optimally within less than 18 minutes.

### 4.3 Global Instruction Scheduling

The ultimate scope of instruction scheduling is to consider whole functions simultaneously, without making any assumption on the CFG structure. This is referred to as *global instruction scheduling* in this survey.

Global instruction scheduling allows an instruction to be moved from its original basic block (called *home basic block*) to any of its predecessor or successor basic blocks (called *candidate basic blocks*). The decision to move an instruction to a predecessor (successor) basic block is called *upward (downward) code motion*. An exception is made for *non-speculative* instructions (for example, those which can trigger exceptions), whose candidate basic blocks are limited to avoid altering the semantics of the original program. Handling compensation code in global scheduling becomes a general problem in which arbitrarily many copies of a moved instruction might need to be replicated in different blocks to compensate for its motion.

The only combinatorial approach to global instruction scheduling that does not make any significant assumption on the CFG structure has been proposed by Winkel [91, 93]. Winkel's IP-based approach is highly ambitious in that it captures 15 different types of code motion proposed in previous literature. However, the basic model is relatively compact, although the complete model comprises 11 types of variables and 35 types of constraints to deal with the idiosyncrasies of the targeted processor and advanced model extensions [93]. The basic model variable types are as follows: *scheduling variables*  $s(i, k, b)$  indicate whether instruction  $i$  is scheduled in cycle  $k$  in basic block  $b$ , *global motion variables*  $c(i, b)$  indicate whether a copy of instruction  $i$  is scheduled on all program paths preceding basic block  $b$ , and *block makespan variables*  $m(b, k)$  indicate whether basic block  $b$  has a makespan of  $k$ . The basic model includes four types of constraints to enforce that every path leading to the home basic block of each instruction  $i$  contains a scheduled copy of  $i$ , the capacity of the resources is not exceeded in any cycle, the data dependencies among instructions are satisfied, and the makespan of each basic block  $b$  is equal to the latest cycle of the instructions scheduled in  $b$ . Additional variables and constraints are proposed, for example, to discard bundles containing incompatible groups

<b>approach</b>	<b>TC</b>	<b>SC</b>	<b>BD</b>	<b>MU</b>	<b>2D</b>	<b>AS</b>	<b>SZ</b>
Arya [6]	IP	local	-	✓	✓	✓	36
Ertl and Krall [23]	CP	local	-	✓	✓	✓	24
Leupers and Marwedel [63]	IP	local	✓	✓	-	-	45
Wilken <i>et al.</i> [88]	IP	local	-	-	-	-	1000
Van Beek and Wilken [87]	CP	local	-	-	-	-	1000
Malik <i>et al.</i> [65]	CP	local	✓	✓	-	-	2600
Malik <i>et al.</i> [64]	CP	superblock	✓	✓	✓	✓	2600
Govindarajan <i>et al.</i> [34]	IP	sw. pipelining	✓	✓	✓	-	?
Altman <i>et al.</i> [4]	IP	sw. pipelining	✓	✓	✓	✓	?
Winkel <i>et al.</i> [93]	IP	global	✓	✓	✓	-	600

Table 3: Combinatorial instruction scheduling approaches: technique (TC), scope (SC), bundling (BD), multiple usage (MU), two-dimensional usage (2D), asynchronous usage (AS), and size of largest optimal solution in number of instructions (SZ).

of instructions [91] and to handle two-dimensional resource usage across basic block boundaries [93]. A detailed account of the model extensions is given in Winkel’s doctoral dissertation [92, Chapter 6]. The objective function is to minimize the weighted makespan of the full function, that is the sum of the makespan of each basic block weighed by its estimated execution frequency.

Winkel’s approach solves the generated IP problems with a commercial IP solver. In contrast to most of the surveyed approaches, Winkel’s model is demonstrated analytically to yield combinatorial problems which can be solved efficiently with IP techniques (that is, problems for which the LP relaxations provide very tight bounds) [92, Chapter 5]. This analytic result is confirmed by a thorough experimental evaluation on 104 functions from SPEC2006, containing up to 600 instructions and comprising 90% of the execution time of the benchmark suite. The experiments for the Itanium 2 processor (with a bundle width of six instructions) show that the approach is feasible for functions with some hundreds of instructions, yielding a runtime speed-up of 10% in comparison with a heuristic global scheduler and 91% in comparison to optimal local instruction scheduling.

## 5 Integrated Approaches

As explained in the introduction, the strong interdependencies between register allocation and instruction scheduling call for integrated approaches where trade-offs can be accurately handled. Combinatorial optimization is well suited for this task since, as opposed to heuristic algorithms, combinatorial models are compositional. Consequently, a significant body of literature has been built on combinatorial integrated approaches. Section 5.1 discusses approaches fo-

ocusing on register allocation and instruction scheduling. Section 5.2 reviews approaches that take one step further, incorporating instruction selection into the combinatorial model.

## 5.1 Register Allocation and Instruction Scheduling

**PROPAN.** In 2000, Kästner proposes an IP-based approach that integrates instruction scheduling with register assignment in the context of *PROPAN*, a post-pass optimization system [50]. PROPAN provides two alternative IP models: a *time-based model* and an *order-based model*.

The *time-based model* captures only instruction scheduling, and it follows the structure of the majority of the instruction scheduling models reviewed in Section 4. The model uses a 0-1 variable for each instruction  $i$ , cycle  $k$ , and resource  $l$  to indicate whether  $i$  is issued in  $k$  and allocated to resource  $l$ ; single-scheduling and dependency constraints similar to those of Leupers and Marwedel (see Section 4.1); and resource constraints to ensure that the capacity of the resources is not exceeded in any cycle.

The *order-based model* is largely based on a model devised by Zhang [94] and later adapted by Kästner and Langenbach [51]. This model departs from the previously introduced IP models for instruction scheduling in that it uses a non 0-1 integer variable to represent the issue cycle of each instruction while retaining the capacity to capture relatively complex resource constraints. As Arya’s model (see Section 4.1), dependency constraints are modeled with inequalities on pairs of cycle variables. Resource constraints are modeled as a resource allocation subproblem based on flow networks, where a resource  $l$  that *flows* through a sequence of instructions represents successive use of  $l$  by these instructions. The 0-1 variables  $u(l, i, j)$  indicate whether resource  $l$  flows from instruction  $i$  to instruction  $j$ . Whenever this is the case, *resource serial constraints* ensure that  $i$  is issued before  $j$ .

Kästner describes an extension of the order-based model to incorporate register assignment. Register assignment is handled in a similar way to resource allocation. A flow network is constructed where registers flow through instructions, where a register flowing through two instructions represents successive writes to that register by these instructions. 0-1 variables  $r(i, j)$  indicate whether a register flows from instruction  $i$  to instruction  $j$ . *Register serial constraints* ensure that this is only possible if  $j$  is scheduled in a cycle in which the temporaries defined by  $i$  are not live anymore.

The model’s scope is the superblock [49, Chapter 7], and the objective function minimizes its makespan. More detail on both time- and order-based models is given in Kästner’s doctoral dissertation [49, Chapter 5].

The experimental results reported by Kästner are limited to nine superblocks of up to 42 instructions, extracted out of digital signal processing applications. The experiments are conducted for two different processors. For the ADSP-2106x processor, PROPAN (with the order-based model) generates code that is almost 20% faster than list scheduling in conjunction with optimal register assignment. For the TriMedia TM1000 processor, both models are compared

after solving the register assignment separately. The results show that, for a processor with a wide bundle like TriMedia TM1000 the time-based model performs better.

**UNISON.** In 2012, Castañeda *et al.* present *UNISON*, a CP approach that integrates multiple subproblems of register allocation with instruction scheduling [10]. Unlike most integrated combinatorial approaches which extend instruction scheduling with one or a few subproblems of register allocation, UNISON focuses on capturing a wide array of them, including register assignment and packing, coalescing, and spilling for multiple register banks. The scope of UNISON’s register allocation is global while instructions are scheduled locally.

Castañeda *et al.* propose two transformations that are applied to a function before formulating a CP problem: *Linear Static Single Assignment construction*, where temporaries whose live ranges span several basic blocks are decomposed into one temporary per basic block and related by a congruence; and *copy extension*, where optional copy operations (*copies* for short) are inserted at each definition and use of a temporary to replicate its value. Copies can be implemented by alternative instructions with copy semantics such as register-to-register moves, loads, and stores; or made inactive by the solver. Memory locations are modeled as individual registers, yielding a compact model where the register to which a temporary is assigned determines its allocation (processor registers or memory), and spill code is reduced to simple copies between registers.

The CP model includes three main types of finite integer variables: *register variables*  $r(t)$  give the register to which temporary  $t$  is assigned, *instruction variables*  $i(o)$  give the instruction that implements operation  $o$  (a special instruction is defined to indicate that  $o$  is inactive), and *scheduling variables*  $s(o)$  give the issue cycle of operation  $o$ .

Register assignment is modeled as a rectangle packing problem where the live range of each temporary yields a rectangle to be packed into a rectangular area with dimensions corresponding to issue cycles and registers, similarly to the approach of Pereira and Palsberg [71]. Coalescing constraints make copies inactive if their source and destination temporaries are assigned to the same register. Operand register constraints enforce that the operands of an operation  $o$  are assigned to registers allowed by the instruction that implements  $o$ . Congruence constraints connect congruent temporaries from different basic blocks by assigning them to the same registers. The instruction scheduling constraints are similar to those in the basic model of Malik *et al.* (see Section 4.1), with the difference that dedicated scheduling constraints such as *cumulative* constraints [1] are used instead of the more general *global cardinality* constraints. Scheduling constraints enable an even more compact model as they can capture two-dimensional usage without the need to resort to additional variables and constraints.

In 2014, Castañeda *et al.* propose an extension of UNISON’s model that captures the spill code optimization problem and improves the scope of coalesc-

ing [11]. The extension is based on the introduction of *alternative temporaries* in the program representation and in the CP model, allowing to choose the temporaries that are used and defined by each operation among different alternative temporaries that hold the same value. In this extension, an operation  $o$  is related to a temporary  $t$  by  $o$ 's operands, to which  $t$  can be *connected*. The improved model includes a new type of variables: *temporary connection* variables  $y_p$  give the temporary that is connected to operand  $p$ . The improved model lifts the congruence constraints from temporaries to operands and includes constraints to enforce that operations are active if their defined temporaries are used by other active operations. The objective function is to minimize the *weighted cost* of the input function:

$$\sum_{b \in B} \text{weight}(b) \times \text{cost}(b) \quad (3)$$

where  $B$  is the set of basic blocks in the function and  $\text{cost}(b)$  is a generic cost function for basic block  $b$ . By combining different definitions of  $\text{weight}(b)$  and  $\text{cost}(b)$ , the objective function can be adapted to optimize for different criteria such as speed or code size.

Castañeda *et al.* introduce two methods to improve the robustness and scalability of UNISON: *presolving*, where the CP problem is reformulated before solving with the addition of implied constraints and bounds; and *decomposition*, where the CP problem is split into a global problem (where the decisions affecting multiple basic blocks are taken), and a local problem per basic block where the internal decisions of each basic block are taken independently.

A thorough experimental evaluation on 55 functions of the MediaBench benchmark (of up to 1000 instructions) for the VLIW Hexagon processor shows that UNISON generates code on average 7% faster than that of a traditional approach (LLVM) and can be easily adapted for code size optimization (the resulting code size is on par with LLVM), the presolving and decomposition methods are essential for UNISON's robustness and scalability, and the improved model yields indeed better solutions than the original model from 2012.

**Other register allocation and instruction scheduling approaches.** Other IP approaches have been proposed besides PROPAN and UNISON that integrate multiple-issue instruction scheduling with subproblems of register allocation for different program scopes.

Chang *et al.* propose in 1997 an early IP model for integrated local instruction scheduling and spilling that includes spill code optimization [13]. The approach has similarities with the initial 2012 model of UNISON (see above) in that input programs are extended with optional store and load instructions after temporary definitions and before temporary uses. The model includes three types of variables: *scheduling variables*  $s(i, k)$  indicate whether instruction  $i$  is scheduled in cycle  $k$ , *live range variables*  $l(t, k)$  indicate whether temporary  $t$  is live in cycle  $k$ , and *activeness variables*  $a(i, k)$  indicate whether the store or load instruction  $i$  is active in cycle  $k$ . Single-scheduling, dependency and resource constraints are defined similarly to other time-based models such as that



of Leupers and Marwedel (see Section 4.1). The number of simultaneous live temporaries in each cycle is constrained to be less or equal than the number of available registers. The model only considers a store (load) instruction as active if it is not scheduled in the same cycle as its predecessor (successor) instruction in the DG. The activeness variables make this distinction explicit. Two alternative objective functions are proposed: to minimize the required number of registers given a fixed makespan, and to minimize the makespan given a fixed number of available registers. Two manual formulations of a basic block with 10 instructions for an ideal processor are compared, with and without spilling. The formulation with spilling takes one order of magnitude more time to be solved to optimality.

Nagarakatte and Govindarajan propose in 2007 an IP model for register allocation and spill code scheduling in the scope of software pipelined loops [66]. The approach assumes a software-pipelined loop kernel with a given initiation interval and computes, if possible, a register allocation, including assignment, spilling, and spill code optimization and scheduling. The IP model includes three types of variables: *register assignment variables*  $a(t, r, k)$  indicate whether temporary  $t$  is assigned to register  $r$  in cycle  $k$ , *store variables*  $s(t, r, k)$  indicate whether temporary  $t$  is stored from register  $r$  to memory in cycle  $k$ , and *load variables*  $l(t, r, k)$  indicate whether temporary  $t$  is loaded from memory to register  $r$  in cycle  $k$ .

The model has several types of linear constraints to ensure that: temporaries are assigned to registers in the cycles in which they are defined and used, temporaries loaded from memory are previously stored, temporaries can only be stored if they are assigned to a register, temporaries are assigned to registers in a certain cycle as a continuation of assignments in previous cycles or if they are just loaded from memory, multiple temporaries are not assigned to the same register, and the capacity of the memory functional units is not exceeded. The model also includes some dominance constraints, for instance to enforce that each temporary is stored at most once, and that temporaries whose value is already contained in some register are not loaded from memory. The objective function is to minimize the spill code (number of stores and loads involved in spilling). The experiments for an idealized processor with a bundle width of 10 instructions on 268 loops from SPEC and Perfect Club benchmarks show that Nagarakatte and Govindarajan’s approach generates, on average, 18% and 13% less spill code than the state-of-the-art heuristics for 32 and 16 registers and avoids increasing the initiation interval in 11% and 12% of the non-trivial loops for 32 and 16 registers. In the worst case of a processor with 32 registers, the IP solver takes on average 78 seconds to solve each loop to optimality.

## 5.2 Full Integration

This section surveys the approaches that tackle the *holy grail* of combinatorial code generation: the addition of instruction selection to combinatorial register allocation and instruction scheduling. Instruction selection transforms a program represented with intermediate operations into an equivalent one that uses

instructions of a certain processor. The task usually involves selecting among different *instruction patterns* (*patterns* for short) that can cover one or more nodes and edges of a processor-independent DG. This task closely interacts with register allocation and instruction scheduling, since the selection of processor instructions imposes constraints on the processor resources and register banks that are used by the program.

**Wilson *et al.*** The first combinatorial approach to fully integrated code generation is proposed by Wilson *et al.* in 1994 [89, 90]. The model is designed with the goal of generating code for DSPs with very limited and irregular register banks such as Motorola’s DSP56001. Remarkably, the IP model captures more register allocation subproblems than many approaches without instruction selection. Unfortunately, experimental results are not publicly available, which has limited the impact of the approach.

Wilson *et al.*’s IP model is rather compact in relation to the amount of captured subproblems. Since instruction selection is included, the input program is represented at a higher level of abstraction, in terms of a DG with processor-independent *operations* and *data edges* (*edges for short*). Both operations as well as edges can be deactivated by the solver to model internal operations of complex processor instructions, optional spilling, and alternative implementations of certain operations such as computations of memory addresses. Unlike most studied approaches, the register allocation submodel is based on edges rather than temporaries, which increases the model granularity since multiple uses of the same temporary yield different edges. Similarly to UNISON (see Section 5.1), Wilson *et al.* extend the input program with optional stores, loads, and register-to-register moves [90].

Similarly to Arya (see Section 4.1), the model uses non 0-1 integer variables  $c_o$  representing the cycle in which each operation  $o$  is issued. As discussed previously, this makes the formulation of dependency constraints straightforward but limits the generality of the resource model. The model is completed with the following types of 0-1 variables: *pattern selection variables*  $s(p)$  indicate whether pattern  $p$  is selected, *operation and edge activation variables*  $a(o)$  and  $a(e)$  indicate whether operation  $o$  and edge  $e$  are active, and *register assignment variables*  $a(e, r)$  indicate whether edge  $e$  is assigned to register  $r$ . Instruction selection constraints enforce that: pattern coverings do not overlap, edges and operations are active iff they are not internalized by a pattern covering, active edges are assigned to a register, edges related across basic blocks are assigned to the same register (similarly to the congruence constraints in UNISON), the definers of active edges are scheduled before their users, active operations that use the same resource are not scheduled in the same cycle, and the definers and users of active edges that are assigned to the same register are scheduled such that the live ranges do not overlap. Besides these constraints, an additional set of constraints allow to model alternative implementations, in which the solver chooses one out of a group of operations to appear in the generated code. The objective function minimizes the weighted makespan of the input function.

approach	TC	SC	SP	RA	CO	SO	RP	LS	RMB	BD	MU	2D	AS	IS	SZ
PROPAN [50]	IP	superblock	-	✓	-	-	-	-	✓	✓	✓	✓	-	-	42
UNISON [11]	CP	global	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	605
Chang <i>et al.</i> [13]	IP	local	✓	-	✓	✓	-	-	✓	✓	✓	-	-	-	$\sim 10$
Nagar. <i>et al.</i> [66]	IP	sw. pipelining	✓	✓	-	✓	-	✓	✓	✓	-	-	-	-	?
Wilson <i>et al.</i> [89]	IP	global	✓	✓	✓	-	-	✓	✓	✓	✓	-	-	✓	30
Gebotys [28]	IP	local	✓	✓	-	✓	-	✓	-	-	✓	-	-	✓	108
ICG [8]	CP	local	✓	✓	✓	✓	-	✓	✓	✓	-	-	-	✓	23
OPTIMIST [21]	IP	sw. pipelining	✓	-	-	✓	-	✓	✓	✓	✓	✓	✓	✓	100

Table 4: Integrated combinatorial approaches: technique (TC), scope (SC), spilling (SP), register assignment (RA), coalescing (CO), spill code optimization (SO), register packing (RP), live-range splitting (LS), rematerialization (RM), multiple register banks (MB), bundling (BD), multiple usage (MU), two-dimensional usage (2D), asynchronous usage (AS), instruction selection (IS), and size of largest optimal solution in number of instructions (SZ).

Unfortunately, Wilson *et al.* do not report detailed experimental results for their integrated code generation approach. The paper discussions indicate that the approach is able to generate optimal code for up to 30 operations, of the same quality as hand-optimized versions. Wilson *et al.* propose some techniques to scale up the approach, including taking preliminary decisions heuristically, and decomposing functions into regions of basic blocks to be solved incrementally [90].

**Gebotys.** An alternative, IP-based approach to fully integrated code generation is due to Gebotys in 1997 [28]. Gebotys’ approach solves instruction compaction as opposed to full instruction scheduling. Instruction compaction assumes a given fixed sequence of instructions and decides, for each instruction  $i$  in the sequence whether  $i$  is scheduled in parallel with its successor. Solving compaction only dramatically reduces the search space at the cost of limiting the capacity to exploit instruction-level parallelism.

The model proposed by Gebotys is composed mostly of linear constraints derived from *Horn clauses*. A Horn clause is a logical formula with at most one conclusion [42], and it is shown that the linear relaxation of an IP problem consisting only of linear constraints derived from Horn clauses gives the optimal solution to the IP problem. Even IP models that are not fully composed of Horn constraints (as is the case for Gebotys’ model) become generally easier to solve, although they must still resort to branch-and-bound in the general case.

As Wilson’s model, Gebotys’ model is based on a DG representation with processor-independent *operations* and *edges*. The model contains two main types of 0-1 variables: *pattern covering variables*  $c(o, p)$  indicate whether pattern  $p$  covers operation  $o$ , and *compaction variables*  $m(o)$  indicate whether operation  $o$  is to be compacted with its successor  $o'$  in the input instruction sequence, due to the same pattern covering both  $o$  and  $o'$ . Instead of defining register assignment variables as in the case of Wilson *et al.*, Gebotys’ model includes patterns for each different flow that a data edge could follow through registers and memory. Thus, selecting a pattern covering implicitly decides the register assignment and possible spilling of the covered edges. This representation is based on the assumption that the target processor has a very small register file, as otherwise the number of patterns would explode. The model include non-Horn constraints to enforce that pattern coverings do not overlap, and Horn constraints to enforce that coverings are selected that assign interfering edges to different registers, and operations are compacted whenever their edge assignments are compatible. The objective function is a weighted sum of three different costs: makespan, code size, and energy consumption (approximated as the number of memory accesses of the generated code).

Gebotys presents experimental results for six single-block signal processing examples and TI’s TMS320C2x – a simple, single-issue DSP with very few registers. In the experiments, the objective function is limited to minimizing the makespan, which in the case of TMS320C2x is directly proportional to code size. The results show a makespan improvement from 9% of up to 118% relative

to the code generated by TI's C compiler. Interestingly, the six basic blocks can be solved to optimality from the first linear relaxation, without resorting to branch-and-bound search. Gebotys also experiments with extending the model with complete instruction scheduling by adding a time dimension to the pattern selection variables, and observes a dramatic increase of the solving time due to an explosion of variables in the model and the insufficiency of the initial linear relaxations to deliver the optimal solution. Since TMS320C2x is a single-issue processor, the extended model does not yield better solutions to any of the six single-block signal processing examples.

**ICG.** Bashford and Leupers propose in 1999 *ICG*, a code generator that integrates instruction selection, register allocation and instruction scheduling using an approach that is partially based on CP [8]. As is common to all fully-integrated approaches, programs in ICG are represented by a DG with abstract operations. Unlike the other approaches, ICG decomposes the solving process into several stages, sacrificing global optimality in favor of solving speed. In Bashford and Leupers's paper, ICG is described as a procedural code generator with limited search capabilities. However, the model is presented in this survey as a monolithic CP model for clarity and consistency.

The ICG model comprises the following variable types: *register variables*  $r(o, op)$  give the register to which each use and definition operand  $op$  of operation  $o$  is assigned, *bundle type variables*  $t(o)$  give the type of bundle to which operation  $o$  belongs, *functional unit variables*  $f(o)$  give the functional unit used by operation  $o$ , and *operation cost variables*  $c(o)$  give the number of issue cycles required to execute operation  $o$ . The register assignment variables include a special register to represent each memory bank. Pattern coverings are not represented explicitly with variables but derived from the remaining decisions. Possible coverings of more than one operation by a pattern  $p$  are represented by special registers assigned to the definition operands covered by  $p$ . The model lacks scheduling variables as instruction scheduling is carried out in a procedural fashion [8, Section 7]. The model contains constraints to ensure that: the registers assigned to operands of different operations are compatible; operations can only be bundled together if they have the same bundle type and do not use the same functional unit; and specific instruction constraints relating register assignments, bundle types, functional units and costs within an operation are satisfied. The remaining constraints necessary for correct code generation such as precedences among operations or non-interference among temporaries are handled by procedural algorithms with a limited form of search.

The solving procedure is split into three steps: first, a constraint problem is formulated and initial propagation is performed to discard invalid registers for operands; then, a procedural, iterative algorithm is run where operations are scheduled and spill code is generated. The latter is performed *on-demand* by extending the problem with new variables and constraints corresponding to load, store, and register-to-register instructions. A certain level of search is permitted within each iteration, and reparation (where the problem is transformed into

an easier one by means of, for example, spilling) is used as a last resort to handle search failures. The (implicit) goals are to minimize makespan and spill code. Finally, search is applied to assign values to all variables that have not been decided yet. A post-processing step is run to handle memory address computations.

Bashford and Leupers evaluate ICG on four basic blocks of the DSPstone benchmark suite for the ADSP-210x processor. The results show that the generated code is as good in terms of makespan as hand-optimized code, and noticeably better than traditionally generated code. However, the scalability of ICG seems limited as generating code for a relatively small basic block of 23 operations takes almost seven seconds, where the immediately smaller basic blocks (17 operations) take less than two seconds.

**OPTIMIST.** The most recent approach to fully integrated code generation is called *OPTIMIST* [21]. This project has explored the application of several search techniques for local code generation, including dynamic programming [53, 54] and genetic algorithms [22]. However, IP has established itself as the technique of choice within OPTIMIST as a natural yet formal method to extend the code generation model to more complex processors [22] and software pipelining [20, 21].

Compared to previous integrated approaches, the OPTIMIST IP model has a rich resource model in which, similarly to the early approach of Ertl and Krall presented in Section 4.1, the resource usage of an instruction is specified per pipeline stage. Another distinctive feature is that copies between different register banks and memory are only modeled implicitly by means of variables indicating their movement, rather than inserted in the input program in a pre-processing stage and defined as optional but otherwise regular operations. Similarly to UNISON (see Section 5.1), memory is modeled as a register bank and thus spilling is unified with handling multiple register banks. The model captures thus register allocation but leaves out register assignment, which precludes coalescing and register packing.

OPTIMIST’s IP model is described first for local integrated code generation and extended later for software pipelining. The model (slightly simplified for clarity) includes the following types of 0-1 variables: *pattern selection variables*  $s(p, k)$  indicate whether pattern  $p$  is selected and issued in cycle  $k$ , *operation and edge covering variables*  $c(p, o, k)$  and  $c(p, e, k)$  indicate whether pattern  $p$  covers operation  $o$  and edge  $e$  in cycle  $k$ , *copy variables*  $x(t, s, d, k)$  indicate whether temporary  $t$  is copied from source register bank  $s$  to destination register bank  $d$  in cycle  $k$ , and *register allocation variables*  $a(t, b, k)$  indicate whether temporary  $t$  is allocated to register bank  $b$  in cycle  $k$ . The model’s instruction selection constraints ensure that all operations are covered by exactly one pattern. The register allocation constraints ensure that the program temporaries: are assigned to the register banks supported by their definer and user instructions in their definition and use cycles; can only be allocated to a register bank  $b$  in a cycle  $k$  if they are defined in cycle  $k$ , are already allocated to  $b$  in cycle

$k - 1$ , or are copied from another register bank in cycle  $k$ ; are not allocated to any register bank if they correspond to internalized edges; and do not exceed the capacity of register banks in any cycle. These register allocation constraints are closely related to earlier IP models discussed in this survey. The first and second constraints are similar to those of Nagarakatte and Govindarajan (see Section 5.1); the third constraint is similar to the second constraint in Wilson *et al.*'s model described above. In both cases, the constraints are lifted from individual registers to register banks. Last, the model's instruction scheduling constraints ensure that: temporaries can only be used and copied after their definition cycle; and resources such as functional units are not overused and the bundle width is not exceeded in any cycle. The objective function, as is common to many local integrated approaches, minimizes the basic block's makespan.

The model extension to single-block software pipelining preserves the same variables and introduces only a few changes in the model constraints. The changes handle two fundamental characteristics of software pipelining: the introduction of cyclic dependencies across loop iterations, and the fact that resource usage by an instruction and temporary live ranges might overlap over time across different iterations. As is common in combinatorial software pipelining, the initiation interval is a fixed parameter and the IP problem is generated and solved for different values until the optimal solution (that which minimizes the initiation interval) is found.

Eriksson *et al.* present experimental results for both the local and the software pipelining IP approaches. The original experimental results are presented in two publications [22, 21]. This survey presents updated results from a rerun reported in Eriksson's doctoral dissertation [19] where a newer IP solver and host architecture are used. The local IP approach is compared to a heuristic code generator based on genetic algorithms (GAs). The input basic blocks are extracted from the *jpeg* and *mpeg2* benchmarks in MediaBench, and the targeted processor is a slightly modified version of TI's C62x. The code generated by the IP approach is better than that generated by the heuristic in around 50% of the cases, where the makespan is improved by around 8 cycles in average and up to 100 cycles in the largest basic block. The largest basic block solved by the IP approach has 191 operations. In the original experiment results published three years before, the largest basic block solved by the IP approach within the same time limit has 52 operations, and only 40% of the basic blocks up to 191 operations can be solved [22, Section 4.2]. This difference illustrates the speed with which solvers and architectures evolve to enable solving increasingly harder combinatorial problems.

The fully-integrated software pipelining IP approach is evaluated by comparing it to a separated approach (where instruction selection and register bank allocation are solved first and the remaining tasks are solved later). The experiments are run for the same processor as the local approach and loop kernels from the SPEC2000, SPEC2006, MediaBench, and Ffmpeg benchmark suites. The results show that the initiation interval of the kernels improved by the integrated approach (which are 39% of the total) is improved by 20% in average [19, Chapter 8]. The approach scales up to loop kernels of approximately

100 operations [19, Chapter 4], 40 operations larger compared to the original experimental results obtained two years before [21].

## 6 Conclusion

This paper has surveyed the existing body of literature on combinatorial register allocation and instruction scheduling, two central compiler back-end problems.

Significant progress has been made for each of the problems in isolation since the application of combinatorial optimization was proposed in the 1980s. Today, the vast majority of register allocation or instruction scheduling problems found in practice can be solved to optimality in the order of seconds. Yet, combinatorial optimization is hardly applied in modern compilers. This is most likely due to the fact that the performance improvements yielded by combinatorial register allocation or instruction scheduling in isolation are not dramatic enough to motivate a paradigm shift in code generation. Circumstantial factors such as licensing mismatches, software integration obstacles, and even cultural differences are also likely to play a part in this lack of adoption.

Integrated code generation has the potential to bring the kind of improvements in code quality that would motivate a paradigm shift, but further research is required in two directions: devising combinatorial models that capture all relevant code generation subproblems to deliver the highest code quality, and developing solving methods that make it scale to the point where most practical problems can be solved in reasonable time.

## Acknowledgments

This research has been partially funded by LM Ericsson AB and the Swedish Research Council (VR 621-2011-6229). The authors are grateful for helpful comments from Sven Mallach, Mattias Eriksson, Mats Carlsson, Bernhard Scholz, and Lang Hames.

## References

- [1] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.
- [2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. 1993.
- [3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, September 1995.
- [4] Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *PLDI*, pages 139–150, 1995.



- [5] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. *SIGPLAN Not.*, 36:243–253, May 2001.
- [6] S. Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Transactions on Computers*, C-34(11):981–995, Nov 1985.
- [7] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling*. 2001.
- [8] Steven Bashford and Rainer Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems*, pages 119–165, March 1999.
- [9] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16:428–455, 1994.
- [10] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. Constraint-based register allocation and instruction scheduling. In *CP*, volume 7514, pages 750–766, 2012.
- [11] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Combinatorial spill code optimization and ultimate coalescing. In *LCTES*, pages 23–32, 2014.
- [12] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.
- [13] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers Math. Applic.*, 34:1–14, November 1997.
- [14] Geoffrey Chu and Peter J. Stuckey. A generic method for identifying and exploiting dominance relations. In *CP*, volume 7514, pages 6–22, 2012.
- [15] Quentin Colombet, Florian Brandner, and Alain Darte. Studying optimal spilling in the light of ssa. In *CASES*, pages 25–34, 2011.
- [16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. MIT, 2009.
- [17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [18] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. Progressive spill code placement. In *CASES '09*, pages 77–86, 2009.

- [19] Mattias Eriksson. *Integrated Code Generation*. PhD thesis, Linköping University, Sweden, 2011.
- [20] Mattias Eriksson and Christoph Kessler. Integrated modulo scheduling for clustered VLIW architectures. In *HiPEAC*, pages 65–79, 2009.
- [21] Mattias Eriksson and Christoph Kessler. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.*, 11S(1):19:1–19:24, June 2012.
- [22] Mattias Eriksson, Oskar Skoog, and Christoph Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *SCOPES*, pages 11–20, 2008.
- [23] M. Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming*, volume 528, pages 75–86, 1991.
- [24] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981.
- [25] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA*, pages 140–150, 1983.
- [26] Eugene C. Freuder. Exploiting structure in constraint satisfaction problems. In *CP*, pages 51–74, 1994.
- [27] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO-35*, pages 245–256, 2002.
- [28] Catherine H. Gebotys. An efficient model for DSP code generation: Performance, code size, estimated energy. In *ISSS*, pages 41–47, 1997.
- [29] Catherine H. Gebotys and Mohamed I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *DAC*, pages 2–7, 1991.
- [30] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18:300–324, 1996.
- [31] Carla P. Gomes and Bart Selman. Algorithm portfolio design: Theory vs. practice. In *UAI*, pages 190–197, 1997.
- [32] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS’88*, pages 442–452, 1988.
- [33] David W. Goodwin and Kent Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Software – Practice and Experience*, 26:929–965, August 1996.
- [34] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. In *Parallel Processing: CONPAR 94 VAPP VI*, volume 854, pages 640–651, 1994.

- [35] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *CC*, volume 4420, pages 111–125. Springer-Verlag, 2007.
- [36] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, volume 3923, pages 247–262, 2006.
- [37] Lang Hames. *Specification driven register allocation*. PhD thesis, University of Sydney, Australia, 2011.
- [38] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *JMLC’06*, pages 346–361, 2006.
- [39] Mark Heffernan and Kent Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8(5):427–451, 2006.
- [40] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th edition, 2011.
- [41] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC’03*, pages 202–213, 2003.
- [42] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 03 1951.
- [43] Cheng-Tsung Hwang, J.-H. Lee, and Yu-Chin Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, April 1991.
- [44] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. 7(1-2):229–248, May 1993.
- [45] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual*, September 2014.
- [46] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.
- [47] Richard K. Johnson. A survey of register allocation. Technical report, Carnegie Mellon University, United States of America, 1973.
- [48] Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice Hall, 1195.
- [49] Daniel Kästner. *Retargetable Postpass Optimisation by Integer Linear Programming*. PhD thesis, Saarland University, Germany, 2000.

- [50] Daniel Kästner. PROPAN: A retargetable system for postpass optimisations and analyses. In *LCTES*, volume 1985, pages 63–80, 2001.
- [51] Daniel Kästner and Marc Langenbach. Code optimization by integer linear programming. In *CC*, volume 1575, pages 122–136, 1999.
- [52] Christoph Kessler. Compiling for VLIW DSPs. In *Handbook of Signal Processing Systems*, pages 603–638. 2010.
- [53] Christoph Kessler and Andrzej Bednarski. A dynamic programming approach to optimal integrated code generation. In *LCTES*, pages 165–174, 2001.
- [54] Christoph Kessler and Andrzej Bednarski. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience*, 18:1353–1390, 2006.
- [55] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO’05*, pages 269–280, 2005.
- [56] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. *SIGPLAN Not.*, 41:204–215, June 2006.
- [57] Timothy Kong and Kent Wilken. Precise register allocation for irregular architectures. In *MICRO-31*, pages 297–307, 1998.
- [58] Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957.
- [59] Philipp Klaus Krause. Optimal register allocation in polynomial time. In *CC*, volume 7791, pages 1–20, 2013.
- [60] Ulrich Kremer. Optimal and near-optimal solutions for hard compilation problems. *Parallel Processing Letters*, 7(4):371–378, 1997.
- [61] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *PLDI*, pages 318–328, 1988.
- [62] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
- [63] Rainer Leupers and Peter Marwedel. Time-constrained code compaction for DSP’s. *Transactions on Very Large Scale Integration Systems*, 5:112–122, March 1997.
- [64] Abid M. Malik, Michael Chase, Tyrel Russell, and Peter Van Beek. An application of constraint programming to superblock instruction scheduling. In *CP*, volume 5202, pages 97–111, 2008.

- [65] Abid M. Malik, Jim McInnes, and Peter Van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International Journal on Artificial Intelligence Tools*, 17(1):37–54, 2008.
- [66] Santosh G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *CC*, volume 4420, pages 126–140, 2007.
- [67] V. Nandivada, Fernando Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis*, volume 4634, pages 153–169, 2007.
- [68] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. 1999.
- [69] Fernando Pereira and Jens Palsberg. Register allocation by puzzle solving. *SIGPLAN Not.*, 43:216–226, June 2008.
- [70] Fernando Magno Quintão Pereira. A survey on register allocation. Technical report, University of California, United States of America, 2008.
- [71] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. pages 216–226, 2008.
- [72] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, 1999.
- [73] Jonathan Protzenko. A survey of register allocation techniques. Technical report, École Polytechnique, France, 2009.
- [74] Vaclav Rajlich and M. Drew Moshier. A survey of algorithms for register allocation in straight-line programs. Technical report, University of Michigan, United States of America, 1984.
- [75] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl.*, 12(4):183–198, December 1981.
- [76] B. Ramakrishna Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. 7:9–50, May 1993.
- [77] Colin R. Reeves. Genetic algorithms. In *Handbook of Metaheuristics*, pages 109–139. 2010.
- [78] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, volume 3258, pages 557–571, 2004.
- [79] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.

- [80] Jean-Charles Régim. Generalized arc consistency for global cardinality constraint. In *AAAI*, pages 209–215, 1996.
- [81] Hongbo Rong and R. Govindarajan. Advances in software pipelining. In *The Compiler Design Handbook*, pages 662–734. CRC, 2nd edition, 2007.
- [82] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. 2006.
- [83] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. *SIGPLAN Not.*, 37:139–148, June 2002.
- [84] Ghassan Shobaki and Kent Wilken. Optimal superblock scheduling using enumeration. In *MICRO-37*, pages 283–293, 2004.
- [85] Ghassan Shobaki, Kent Wilken, and Mark Heffernan. Optimal trace scheduling using enumeration. 5:19:1–19:32, March 2009.
- [86] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, May 1998.
- [87] Peter Van Beek and Kent Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *CP*, volume 2239, pages 625–639, 2001.
- [88] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *SIGPLAN Not.*, 35:121–133, May 2000.
- [89] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. An integrated approach to retargetable code generation. In *ISSS*, pages 70–75, 1994.
- [90] Tom Wilson, Gary Grewal, Shawn Henshall, and Dilip Banerji. An ILP-based approach to code generation. In *Code Generation for Embedded Processors*, pages 103–118. 2002.
- [91] Sebastian Winkel. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *CGO’04*, pages 189–200, 2004.
- [92] Sebastian Winkel. *Optimal global instruction scheduling for the Itanium processor architecture*. PhD thesis, Saarland University, Germany, 2004.
- [93] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *MICRO-40*, pages 43–55, 2007.
- [94] Li Zhang. *Scheduling and Allocation with Integer Linear Programming*. PhD thesis, Saarland University, Germany, 1996.